# 设计原则

## 单一职责原则

就类而言，仅有一个引起该类变化的原因。 不要把界面处理，算法解析，数据库访问放入同一个类中。 不便维护及复用。

## 开放-封闭原则

软件实体(模块、类、函数)可以拓展， 但是不能修改。 易维护， 不容易出问题。

## 依赖倒转原则

高层模块不能依赖底层模块， 底层模块也不能依赖高层模块;抽象不依赖细节， 细节依赖抽象。如主体(高层)调用数据库访问函数，但不能依赖数据库访问函数(底层)， 因为可能主体要使用不同的数据库方式。

# 单例模式(Single)

构造析构函数为protect(不用private是为了可以继承)，使其该类不能通过外部(主要是禁止外部栈对象)来调用构造函数创建对象。

应用:角色创建或代替全局对象；防止不同线程多次初始化，一般在主线程处进行初始化。

饿汉模式：main函数之前创建对象(new或者static)。(线程安全) 需要注意对象初始化顺序问题。

懒汉模式：调用getInstance时创建对象(new或者static)。(线程不安全，需要加锁)。

## 代码示例

示例一(饿汉) :

头文件:

Class A

{

public:

static A& getInstance(){return m\_sInstance;} //返回引用

private:

A(){}

~A()

static A m\_sInstance; };

源文件: A A::m\_sInstance; //系统自动分配到静态存储区。

示例二 (懒汉):

头文件:

Class A

{

public:

static A& getInstance()

{

//Lock();

static A m\_sInstance; //C++11标准不用加锁，线程安全， 否则需要加锁。

//unLock();

return m\_sInstance;//返回引用

} private:

A(){}

~A()

# 工厂模式(Factory)

原理:工厂模式主要是用来封装实例化的模式，通过成员函数封装new的各种操作来提供客户程序使用,比如将实例化对象以及初始化等工作封装到这个成员函数中。

应用: 在new对象等初始化实例操作时， 可考虑工厂模式。

## 代码示例

class Product

{

public:

virtual ~Product() = 0;

protected:

Product(){}

};

class ConcreteProduct : public Product

{

public:

ConcreteProduct(){}

~ConcreteProduct(){}

};

class Factory

{

public:

virtual ~Factory() = 0;

virtual Product\* CreateProduct() = 0;

protected:

Factory(){}

};

class ConcreteFactory : public Factory

{

public:

~ConcreteFactory(){}

ConcreteFactory(){}

Product\* CreateProduct() {return new ConcreteProduct()}

};

int main(int argc, char\* argv[])

{

Factory \*ct = new ConcreteFactory();

Product \* pt = ct->CreateProduct();

if (ct)

delete ct;

if (pt)

delete pt;

return 0;

}

# 抽象工厂模式(AbstractFactory)

原理:当抽象产品类有多个子产品类时， 工厂模式不清楚要选择哪个子产品类， 便可以使用两个工厂类来分别选择生成不同的产品， 但不如索性将这两个工厂类抽象出来， 即使用抽象工厂模式，每个子工厂模式对应一个具体的子产品类。

## 代码示例

class AbstractEnemyFactory

{protected:

AbstractEnemyFactory();

public:

~AbstractEnemyFactory();

virtual Soldier\* MakeSoldier() = 0; //士兵类

virtual Monster\* MakeMonster() = 0; //怪物

virtual SuperMonster\* MakeSuperMonster() = 0; //超级怪兽};

class EasyLevelEnemyFactory : public AbstractEnemyFactory

{public:

Soldier\* MakeSoldier(){ return new SillySoldier();}

Monster\* MakeMonster(){ return new SillyMonster();}

SuperMonster\* MakeSuperMonster() {return new SillySuperMonster();} };

class DieHardLevelEnemyFactory : public AbstractEnemyFactory

{public:

Soldier\* MakeSoldier(){ return new BadSoldier();}

Monster\* MakeMonster(){ return new BadMonster();}

SuperMonster\* MakeSuperMonster() {return new BadSuperMonster();}};

class GameApp

{public:

void fun()

{if (bool)

m\_factory = new EasyLevelEnemyFactory();

else

m\_factory = new DirHardLevelEnemyFactory();

}

private:

AbstractEnemyFactory\* m\_factory; //这是抽象工厂的优势，应用程序把m\_factory视为唯一对象生成者。};

# 桥接模式(Bridge)

## 介绍

好处:

使用组合（委托）的方式将抽象和实现彻底地解耦，这样的好处是抽象和实现可以分别独立地变化，系统的耦合性也得到了很好的降低。

比如在抽象部分的子类的函数中添加功能函数，可使不同实现部分的所有子类都拥有该功能。

## 使用场景

场景一:

1）客户给了你一个需求，于是使用一个类来实现（A）；

2）客户需求变化，有两个算法实现功能，于是改变设计，我们通过一个抽象的基类，再定义两个具体类实现两个不同的算法（A1和A2）；

3）客户又告诉我们说对于不同的操作系统，于是再抽象一个层次，作为一个抽象基类A0，在分别为每个操作系统派生具体类（A00和A01，其中A00表示原来的类A）实现不同操作系统上的客户需求，这样我们就有了一共4个类。

4）可能用户的需求又有变化，比如说又有了一种新的算法……..

5）我们陷入了一个需求变化的郁闷当中，也因此带来了类的迅速膨胀。

场景二:

抽象部分可以用于:界面部分

实现部分可以用于:对数据的处理

## 代码示例

示例一:

class AbstractionImp

{

public:

virtual ~AbstractionImp(){}

virtual void Operation() = 0;

protected:

AbstractionImp(){}

};

class ConcreteAbstractionImpA : public AbstractionImp

{

public:

ConcreteAbstractionImpA(){}

virtual ~ConcreteAbstractionImpA(){}

void Operation(){cout << "ConcreteAbstractionImpA" << endl;}

};

class ConcreteAbstractionImpB : public AbstractionImp

{

public:

ConcreteAbstractionImpB(){}

virtual ~ConcreteAbstractionImpB(){}

void Operation(){cout << "ConcreteAbstractionImpB" << endl;}

};

class Abstraction

{

public:

virtual ~Abstraction(){}

virtual void Operation() = 0;

protected:

Abstraction(){}

};

class RefinedAbstraction : public Abstraction

{

public:

RefinedAbstraction(AbstractionImp\* imp){m\_imp = imp;}

virtual ~RefinedAbstraction(){}

void Operation(){m\_imp->Operation();} //若在该函数内添加额外功能：就是Decorator(修饰)模式

private:

AbstractionImp \* m\_imp;

};

int main()

{

//使抽象部分和实现部分分离

AbstractionImp \*imp = new ConcreteAbstractionImpA(); //实现部分

Abstraction \*ab = new RefinedAbstraction(imp); //抽象部分,

ab->Operation();

delete ab;

delete imp;

return 0;

}

# 装饰模式(Decorator)

原理：不改动原类代码，新增的类中保存传进来的原类的指针，从而保证了“开放-封闭则。

应用: 原类的核心代码需要修饰时。 比如增加的辅助功能等。

优点: 简化了原类的代码，核心功能与辅助功能解耦。

## 代码示例

示例一:

class Component //相对于电池测试

{

public:

virtual ~Component();

virtual void Operation() = 0;

protected:

Component();

};

class ConcreteComponent : public Component //电池测试的版本

{

public:

ConcreteComponent();

virtual ~ConcreteComponent();

void Operation();

};

class Decorator : public Component //电池测试每个不同版本的额外功能

{

public:

Decorator(Component\* com);

virtual ~Decorator();

void Operation();

protected:

Component\* m\_com;

};

class ConcreteDecorator : public Decorator

{

public:

ConcreteDecorator(Component\* com);

virtual ~ConcreteDecorator();

void Operation();

void AddedBehavior();

};

Component::Component()

{

cout << "Component" << endl;

}

Component::~Component()

{

cout << "~Component" << endl;

}

ConcreteComponent::ConcreteComponent()

{

cout << "ConcreteComponent" << endl;

}

ConcreteComponent::~ConcreteComponent()

{

cout << "~ConcreteComponent" << endl;

}

void ConcreteComponent::Operation()

{

cout << "ConcreteComponent operation" << endl;

}

Decorator::Decorator(Component\* com)

{

m\_com = com;

}

Decorator::~Decorator()

{

cout << "~Decorator" << endl;

delete m\_com;

}

void Decorator::Operation()

{

cout << "Decorator operation" << endl;

}

ConcreteDecorator::ConcreteDecorator(Component\* com) : Decorator(com)

{

}

ConcreteDecorator::~ConcreteDecorator()

{

}

void ConcreteDecorator::AddedBehavior()

{

cout << "AddedBehavior" << endl;

}

void ConcreteDecorator::Operation()

{

m\_com->Operation();

AddedBehavior();//为每个不同的Component子类增加额外功能。

}

int main()

{

/\*为ConcreteComponent类添加一项ConcreteDecorator类的额外功能\*/

Component \*com = new ConcreteComponent();

Decorator \*dec = new ConcreteDecorator(com); //Decorator这一个类可以维护更多Component的子类(特点)

dec->Operation();//Decorator的多态是解决增加不同的额外功能,只需增加1种额外功能则不需要ConcreteDecorator的基类。

delete dec; //释放dec和com

return 0;

}

# 观察者模式(Observer)

原理: 主题类列表变量中存放多个观察者类，更新变化时调用每个观察者的相同抽象函数。 而每个观察者类中都可存放主题类指针获取相同主题类中的变量。

应用: 当一个对象改变需要改变其他对象时。如对同一组数据进行统计分析时候，我们希望能够提供多种形式的表示,百分比，柱状图等。

优点: 解除耦合， 使双方的耦合依赖于抽象，并非依赖于具体类。

## 代码示例

示例一:

class Observer;

class Subject

{

public:

virtual ~Subject(){}

virtual void Attach(Observer\* obv){m\_listObvs->push\_front(obv);}

virtual void Detach(Observer\* obv){if(obv != NULL)m\_listObvs->remove(obv);}

virtual void Notify();

virtual void SetState(const string& str) = 0;

virtual string GetState() = 0;

protected:

Subject(){m\_listObvs = new list<Observer\*>;}

private:

list<Observer\*> \*m\_listObvs;

};

void Subject::Notify()

{

list<Observer\*>::iterator it;

for (it = m\_listObvs->begin(); it != m\_listObvs->end(); ++it)

{

(\*it)->Update(this);

}

}

class ConcreteSubject : public Subject

{

public:

ConcreteSubject(){m\_str = '0';}

~ConcreteSubject();

string GetState(){return m\_str;}

void SetState(const string &str){m\_str = str;}

private:

string m\_str;

};

class Observer

{

public:

virtual ~Observer();

virtual void Update(Subject \*sub) = 0;

virtual void PrintInfo() = 0;

protected:

Observer(){m\_str = '0';}

string m\_str;

};

class ConcreteObserverA : public Observer

{

public:

virtual Subject\* GetSubject(){return m\_sub;}

ConcreteObserverA(Subject\* sub){m\_sub = sub; m\_sub->Attach(this);}

virtual ~ConcreteObserverA(){m\_sub->Detach(this); if (m\_sub != 0)delete m\_sub;}

void Update(Subject\* sub){m\_str = sub->GetState(); PrintInfo();}

void PrintInfo(){cout << "ConcreteObserverA" << m\_sub->GetState() << endl;}

private:

Subject \*m\_sub;

};

class ConcreteObserverB : public Observer

{

public:

virtual Subject\* GetSubject(){return m\_sub;}

ConcreteObserverB(Subject\* sub){m\_sub = sub; m\_sub->Attach(this);}

virtual ~ConcreteObserverB(){m\_sub->Detach(this); if (m\_sub != 0)delete m\_sub;}

void Update(Subject\* sub){m\_str = sub->GetState(); PrintInfo();}

void PrintInfo(){cout << "ConcreteObserverB" << m\_sub->GetState() << endl;}

private:

Subject \*m\_sub;

};

int main()

{

Subject\* sub = new ConcreteSubject();

Observer\* o1 = new ConcreteObserverA(sub);//将需要同步的类，放入 Sub类的容器中。

Observer\* o2 = new ConcreteObserverB(sub);

sub->SetState("old");

sub->Notify();//更改所有同步的类的数据。

sub->SetState("new");

sub->Notify();

return 0;

}

# 状态模式(State)

原理: 主类中增加了状态父类的指针，客户端初始一个状态类指针传给主类，主类函数调用其状态类的功能函数。 状态类的功能函数根据条件判断是否在调用主类函数进行状态切换,否则实现该状态的功能。 状态类中一般不放入属性。

应用: 早、中、晚时间段分别做啥事。

优点: 解决了switch等庞大分支结构， 将判断操作分离到每个状态类中。

## 代码示例

示例一:

class Context;

class State //状态类一般不放属性，只放方法。

{

public:

State(){}

virtual ~State(){}

virtual void OperationInterface(Context\* con) = 0;

virtual void OperationChangeState(Context\* con) = 0;

protected:

bool ChangeState(Context\* con, State\* st){con->ChangeState(st);return true;}

};

class ConcreteStateA : public State

{

public:

ConcreteStateA(){}

virtual ~ConcreteStateA(){}

virtual void OperationInterface(Context\* con){cout << "ConcreteStateA" << endl;}

virtual void OperationChangeState(Context\* con){OperationInterface(con);ChangeState(con, new ConcreteStateB()/\*状态改变\*/);}

};

/\*State及其子类中的操作都将Context\*传入作为参数，

其主要目的是State类可以通过这个指针调用Context中的方法（在本示例代码中没有体现）。

这也是State模式和Strategy模式的最大区别所在\*/

class ConcreteStateB : public State

{

public:

ConcreteStateB(){}

virtual ~ConcreteStateB(){}

virtual void OperationInterface(Context\* con){cout << "ConcreteStateB" << endl;}//利用con调该context类方法

virtual void OperationChangeState(Context\* con){OperationInterface(con); ChangeState(con, new ConcreteStateA());}

};

class Context

{

public:

Context(){}

Context(State\* st){m\_state = st;}

~Context(){delete m\_state;}

virtual void OperationInterface(){m\_state->OperationInterface(this);}

virtual void OperationChangeState(){m\_state->OperationChangeState(this);}

private:

friend class State;

bool ChangeState(State\* st){m\_state = st;return true;}

private:

State\* m\_state;

};

int main()

{

State\* st = new ConcreteStateA();

Context\* con = new Context(st);

con->OperationChangeState();

con->OperationChangeState();

con->OperationChangeState();

if (con != NULL) delete con;

if (st != NULL) st = NULL;

return 0;

}

# 中介模式(Mediator)

## 介绍

好处:

Mediator模式提供将对象间的交互和通讯封装在一个类中，各个对象间的通信不必显势去声明和引用，大大降低了系统的复杂性能和耦合性，以及集中管理。

## 使用场景

场景一:系统对象过多，不变管理。

## 代码示例

示例一:

class Mediator;

class Colleage

{

public:

virtual ~Colleage(){}

virtual void Action() = 0;

virtual void SetState(const string &sdt) = 0;

virtual string GetState() = 0;

protected:

Colleage();

Colleage(Mediator \*mdt) : m\_mdt(mdt){}

Mediator \*m\_mdt;

};

class ConcreteColleageA : public Colleage

{

public:

ConcreteColleageA(){}

ConcreteColleageA(Mediator \*mdt) : Colleage(mdt){}

~ConcreteColleageA(){}

void Action(){m\_mdt->DoActionFromAtoB(); cout << GetState() << endl;}

void SetState(const string &sdt){m\_sdt = sdt;}

string GetState(){return m\_sdt;}

private:

string m\_sdt;

};

class ConcreteColleageB : public Colleage

{

public:

ConcreteColleageB(){}

ConcreteColleageB(Mediator \*mdt) : Colleage(mdt){}

~ConcreteColleageB(){}

void Action(){m\_mdt->DoActionFromBtoA(); cout << GetState() << endl;}

void SetState(const string &sdt){m\_sdt = sdt;}

string GetState(){return m\_sdt;}

private:

string m\_sdt;

};

class Mediator //该类负责管理其他对象的操作，而每个被管理的类，都需要包含该管理类。

{

public:

virtual ~Mediator(){}

virtual void DoActionFromAtoB() = 0;

virtual void DoActionFromBtoA() = 0;

protected:

Mediator(){}

};

class ConcreteMediator : public Mediator

{

public:

ConcreteMediator(){}

ConcreteMediator(Colleage \*clgA, Colleage \*clgB) : m\_clgA(clgA), m\_clgB(clgB){}

~ConcreteMediator(){}

void SetConcreteColleageA(Colleage \*clgA){m\_clgA = clgA;}

void SetConcreteColleageB(Colleage \*clgB){m\_clgB = clgB;}

Colleage\* GetConcreteColleageA(){ return m\_clgA;}

Colleage\* GetConcreteColleageB(){ return m\_clgB;}

void IntroColleage(Colleage\* clgA, Colleage\* clgB){m\_clgA = clgA; m\_clgB = clgB;}

void DoActionFromAtoB(){m\_clgB->SetState(m\_clgA->GetState());} //操作被管理的对象的行为

void DoActionFromBtoA(){m\_clgA->SetState(m\_clgB->GetState());} //操作被管理的对象的行为

private:

Colleage\* m\_clgA; //可以是不同被管理的对象的指针。

Colleage\* m\_clgB;

};

int main()

{

ConcreteMediator\* m = new ConcreteMediator();

ConcreteColleageA\* c1 = new ConcreteColleageA(m);

ConcreteColleageB\* c2 = new ConcreteColleageB(m);

m->IntroColleage(c1,c2);

c1->SetState("old");

c2->SetState("old");

c1->Action(); //将c1的string赋值给c2的string值

c2->Action();

c1->SetState("new");

c1->Action();

c2->Action();

c2->SetState("old");

c2->Action();

c1->Action();

return 0;

}

# 策略模式(Strategy)

## 介绍

思路：使用何种策略不交给客户程序判断，不同策略类覆盖虚函数(策略),根据不用类型选择生成不同策略对象,调用该虚函数。

优点:

1）“黑盒”复用，因为被包含对象的内部细节对外是不可见的；

2）封装性好，原因为1）；

3）实现和抽象的依赖性很小（组合对象和被组合对象之间的依赖性小）；

4）可以在运行期间动态定义实现（通过一个指向相同类型的指针，典型的是抽象基类的指针）。

缺点:

1）系统中对象过多。

## 使用场景

面向对象的设计中的有一条很重要的原则就是:优先使用组合。

## 代码示例

示例一:

class Strategy

{

public:

Strategy(){}

virtual ~Strategy(){}

virtual void AlgorithmInterface() = 0;

};

class ConcreteStrategyA : public Strategy

{

public:

ConcreteStrategyA(){}

virtual ~ConcreteStrategyA(){}

void AlgorithmInterface(){cout << "ConcreteStrategyA" << endl;}

};

class ConcreteStrategyB : public Strategy

{

public:

ConcreteStrategyB(){}

virtual ~ConcreteStrategyB(){}

void AlgorithmInterface(){cout << "ConcreteStrategyB" << endl;}

};

class Context

{

public:

Context(){}

Context(Strategy\* stra):m\_stra(stra){}

virtual ~Context(){}

void DoAction(){m\_stra->AlgorithmInterface();}

private:

Strategy \* m\_stra;

};

int main()

{

Strategy\* stra = new ConcreteStrategyA();

Context \* con = new Context(stra);

con->DoAction();

delete con;

delete stra;

return 0;

}

# 原型模式(Prototype)

## 介绍

## 使用场景

## 代码示例

示例一:

class Prototype

{

public:

virtual ~Prototype();

virtual Prototype\* Clone() const = 0;

protected:

Prototype();

};

Prototype::Prototype()

{

cout << "Prototype" << endl;

}

Prototype::~Prototype()

{

cout << "~Prototype" << endl;

}

class ConcretePrototype : public Prototype

{

public:

ConcretePrototype();

ConcretePrototype(const ConcretePrototype& cp);

~ConcretePrototype();

Prototype \* Clone() const;

};

ConcretePrototype::ConcretePrototype()

{

cout << "ConcretePrototype" << endl;

}

ConcretePrototype::~ConcretePrototype()

{

cout << "~ConcretePrototype" << endl;

}

ConcretePrototype::ConcretePrototype(const ConcretePrototype& cp)

{

cout << "ConcretePrototype(const ConcretePrototype&)" << endl;

}

Prototype\* ConcretePrototype::Clone() const

{

return new ConcretePrototype(\*this); //用已有对象(this)创建一个新对象

}

int main()

{

Prototype\* p = new ConcretePrototype();

Prototype\* p1 = p->Clone();

if (p)

{

delete p;

}

if (p1)

{

delete p1;

}

return 0;

}

# 享元模式(Flyweight)

## 介绍

Flyweight模式在实现过程中主要是要为共享对象提供一个存放的“仓库”（对象池），

这里是通过C++ STL中Vector容器，当然就牵涉到STL编程的一些问题（Iterator使用等）。另外应该注意的就是对对象“仓库”（对象池）的管理策略（查找、插入等），这里是通过直接的顺序遍历实现的，当然我们可以使用其他更加有效的索引策略，例如Hash表的管理策略，

## 使用场景

场景一: 一个应用程序使用了太多的对象，就会造成很大的存储开销。特别是对于大量轻量级（细粒度）的对象。

## 代码示例

示例一:

class Flyweight

{

public:

virtual ~Flyweight(){}

virtual void Operation(const string &extrinsicState) = 0;

string GetIntrinsicState() {return m\_intrinsicState;}

protected:

Flyweight(string intrinsicState){m\_intrinsicState = intrinsicState;}

private:

string m\_intrinsicState;

};

class ConcreteFlyweight : public Flyweight

{

public:

ConcreteFlyweight(string intrinsicState):Flyweight(intrinsicState){cout << "ConcreteFlyweight " << endl;}

~ConcreteFlyweight(){}

void Operation(const string& extrinsicState) //该函数用来区分同一个对象不同区别。

{

GetIntrinsicState();//对象池内部的

cout << extrinsicState << endl; //对象池外面的

}

};

class FlyweightFactory

{

public:

FlyweightFactory(){}

~FlyweightFactory(){}

Flyweight\* GetFlyweight(const string &key);

private:

vector<Flyweight\*> m\_fly;

};

Flyweight\* FlyweightFactory::GetFlyweight(const string &key)

{

vector<Flyweight\*>::iterator it = m\_fly.begin();

for(it; it != m\_fly.end(); ++it)

{

if ((\*it)->GetIntrinsicState() == key)

{

cout << "存在则使用" << endl;

return \*it;

}

}

//不存在则创建

Flyweight \*fn = new ConcreteFlyweight(key);

m\_fly.push\_back(fn);

return fn;

}

int main()

{

FlyweightFactory \*fc = new FlyweightFactory();

Flyweight \*fw1 = fc->GetFlyweight("hello"); //fw1和fw3共享同一个对象。

Flyweight \*fw2 = fc->GetFlyweight("world");

Flyweight \*fw3 = fc->GetFlyweight("hello");

return 0;

}

# 适配器模式(Adapter)

## 介绍

优点：1.无需修改原有代码就能引入新功能。

缺点：过多使用适配器，会让系统非常零乱，如果不是很有必要，则不要使用而是直接对系统进行重构。

## 使用场景

复用原有的代码，来实现新的功能时， 一般不考虑在类设计之初使用， 而是双方已有代码不愿修改时，增加适配类来对接调用。

## 代码示例

Class Target //目标类

{

Public :

virtual void Request(){print(“普通请求”);}

}

Class Adaptee //需要适配的类  
{

Public:

Void SpecificRequest(){print(“特殊请求”)}

}

Class Adaptet : public Target //适配器类

{

Public:  
 virtual void Request() //外部调用该函数后，内部转换适配其他接口(a.SpecifiRequest)

{

a.SpecificRequest();

}

Private:

Adaptee a;

}

//客户程序

Int Main()

{

Target\* ta = new Adapter();

Ta->Request();

}

## 介绍

## 使用场景

## 代码示例